The Anti-Goldilocks Debugger

Helping the Average Bear Debug Transparently Transformed Programs

Myoungkyu Song and Eli Tilevich

Dept. of Computer Science
Virginia Tech, Blacksburg, VA 24061, USA
{mksong,tilevich}@cs.vt.edu

Abstract

The practice of enhancing the bytecode of Plain Old Java Objects (POJOs) with additional capabilities, including persistence, distribution, and security, has become an indispensable part of enterprise software development. The resulting transparently-applied, large-scale structural changes to the bytecode significantly complicate symbolic debugging. This demonstration will showcase the Anti-Goldilocks Java (AGJ) debugger, which enables the programmer to trace and debug transparently transformed programs, without the distraction of the bytecode-level enhancements obfuscating the program's source code. AGJ executes a structurally-enhanced program, while dynamically reinterpreting the debugging output (e.g., 'step', 'print variable', etc.) to display program information as pertaining to the original version of the code. AGJ is based on a new debugging architecture that leverages our domain-specific language for describing enhancements.

A paper in the main technical program of OOPSLA 2009 [5] describes the design rationale and implementation details of AGJ. This demonstration will showcase the functionality of our reference implementation by using it to locate bugs in a framework-based enterprise application from the financial industry. Using the domain of transparent persistence, this demonstration will compare AGJ to the standard JDK debugger, thereby highlighting the capabilities of AGJ to cut through the morass of transparent bytecode enhancements in order to find obscure bugs.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—debugging aids, tracing

General Terms Languages, Design, Experimentation

Keywords Debugging, program transformation, bytecode enhancement

1. Introduction

Modern enterprise frameworks enable the programmer to build business logic components using Plain Old Java Objects (POJOs)—application objects that do not implement special interfaces or call framework API methods. POJO-based frameworks have become mainstream in the enterprise Java community, as they improve separation of concerns, speed up development, and improve portability [3].

To provide services to a POJO, enterprise frameworks commonly enhance its bytecode, either statically, as an extra build step, or dynamically, at class load time. A typical bytecode enhancement constitutes a structural transformation that adds methods and fields, changes direct field accesses with setter/getter methods, adds new super classes/interfaces, etc. As a consequence of such bytecode enhancements, the running version of an enterprise application contains functionality that has no representation at the source code level, making the source-level debugging of enterprise applications containing enhanced bytecode nontrivial. Enhancements play an essential role in the architecture of an enterprise application, as they enable POJOs to interact with a framework—one cannot simply turn off the enhancements to make the debugging process easier. As a result, tracing, analyzing, and fixing buggy programs with enhanced bytecode presents a challenge exacerbating the development of framework-based applications.

Standard debuggers fall short when used to debug programs containing enhanced bytecode, as they attempt to show both the original logic and the transparently introduced enhancements, even though they have no source level representation. When it comes to debugging, bytecode enhancement stealthily obfuscates the program, often making it impossible to map the debugged version back to the original source code. The programmer debugging a program that contains enhanced POJOs can feel utterly confused and frustrated. Indeed, seeing your code having been stealthily modified by some external entity conjures up feelings similar to that experienced by a proverbial bear from "Goldilocks and the Three Bears" [1], with the programmer tempted to scream in utter frustration: "Someone's changed my POJO!"

2. Anti-Goldilocks Debugger

To help debug transparently-enhanced programs, we have created a new debugging architecture that augments a standard debugger with the functionality required to dynamically reinterpret the source code information pertaining to enhancements. Specifically, the debugger, on demand, symbolically undoes the enhancements that have been made to the debugged code. Our implementation, showcased in this demonstration, is called Anti-Goldilocks Java or AGJ for short. The intuition behind the name is that we aim at counteracting the actions of Goldilocks, who has meddled with the bears' belongings, surreptitiously altering them. By analogy, Goldilocks is a transparent bytecode enhancer.

As a fully-functional debugger, AGJ can step through the code, set breakpoints, and print variable values. AGJ leverages the Java Platform Debugger Architecture (JPDA)[6], adding special translation modules to the standard layers of protocols and interfaces provided by the JVM. AGJ uses a special purpose domain-specific language, called Structural Enhancements Rules (SER), for documenting bytecode enhancements. By interpreting SER scripts, AGJ dynamically maps the executed enhanced code to the programmer written code, undoing the enhancements on demand.

Figure 1 demonstrates the AGJ architecture, which integrates a SER interpreter. To debug the enhanced bytecode, AGJ takes as input a SER script declaratively describing the enhancements, using it to transform the debugging output reported to the programmer, as if the original version of the code were being executed.

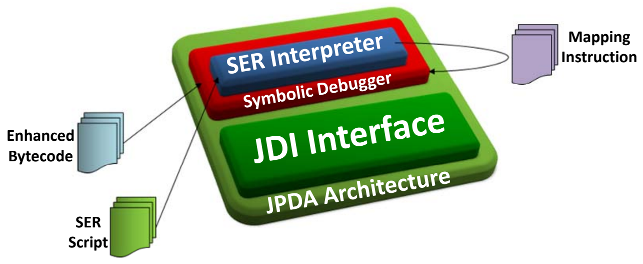

Figure 1. AGJ Architecture.

3. Demonstration Plan

From the programmer's perspective, AGJ is a plug-in replacement for the standard JDK command-line debugger, providing the capabilities to step through the code, set breakpoints, print variable values, etc.

In this demonstration, the utility of AGJ will be shown by tracking bugs in framework applications that use bytecode enhancement as part of their software development cycle. Example applications will include the enhancements used by commercial enterprise frameworks and the ones used in research prototypes.

The demonstration will start with a quick overview of the Structural Enhancements Rules (SER) language, which is used for expressing bytecode enhancements. SER is a declarative, domain-specific language we have created. We will briefly walk through the basic building blocks of SER and show how this language can be used to express various bytecode enhancement strategies.

Then, the main part of the demonstration will consist of using AGJ to find several seeded bugs in two types of applications. The seeded bugs will concern the business logic of the applications, including throwing a `NullPointerException` and using incorrect calculations.

The first application calculates mortgage eligibility and could be used by a bank. This application implements its persistence functionality using the JDO framework [4], which statically enhances the persisted classes to enable them to interact with the framework's runtime. The second application is the "remoting" enhancement used in prior research projects [2, 7] to provide remote access to a class by means of distribution middleware.

For both applications, we will first show the programmer's experience when trying to locate the bug using the standard JDK debugger. Then we will demonstrate how AGJ makes it easier to find the bug, by undoing the JDO framework's enhancements, which complicate the debugging process. We aim at demonstrating how AGJ has the potential to become an effective aid in locating bugs in enhanced programs.

References

[1] A. C. Elms. "The Three Bears": Four interpretations. *The Journal of American Folklore*, 90(357):257–273, 1977.

[2] M. Philippsen and M. Zenger. JavaParty–transparent remote objects in Java. *Concurrency Practice and Experience*, 9(11):1225–1242, 1997.

[3] C. Richardson. Untangling enterprise Java. *ACM Queue*, 4(5):36–44, 2006.

[4] C. Russell. Java Data Objects 2.1, June 2007. `http://db.apache.org/jdo/specifications.html`.

[5] M. Song and E. Tilevich. Enhancing source-level programming tools with an awareness of transparent program transformations. In *OOPSLA '09: Proceedings of the 24th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2009.

[6] Sun Microsystems. Java Platform Debugger Architecture. `http://java.sun.com/javase/technologies/core/toolsapis/jpda/`.

[7] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java application partitioning. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 178–204. Springer-Verlag, LNCS 2374, 2002.